

Flexible Computational Environments for Model Calculations in Particle Physics

Gerard Jungman

January 2000

Abstract

I describe some ideas for creating tools to simplify calculations in particle physics models. Most statements made apply in a more general context. The goal is to create a system which is flexible enough to support exploratory computations, where the methods and strategies implemented could vary as late as run-time; I will discuss what is meant by this. This goal is not profound and could certainly be achieved in the near term; therefore, the problem appears to be more sociological than technical. I also make some comments about the role of simplified and standardized environments in streamlining research communication.

1 The Problem

For purposes of this discussion we can define “calculation” by example; the specific examples I have in mind are generic computations of typical particle properties, such as cross-sections and branching ratios, computations of cosmological interest, and computations of more detailed type, including Monte Carlo event generation. The flavor of all these calculations is that one “has a given model”, specified by a particle spectrum and couplings, and needs to “make predictions for experiments”, where the details of what calculations are to be done, how they are to proceed, etc., may be decided as late as run-time and should not affect the overall organization of the system. We might call this sort of task “plot-making physics”, since the visible end product is often a set of plots which can be used in reports to other groups and funding agencies.

Typically, the need for such calculations is greatest in small experimental groups, who find it necessary to calculate properties of models for comparison with data, but who do not have experts or large support teams on hand to provide needed technical information. In such situations there tend to be long and costly startup times involved in building expertise locally, typically in the form of graduate students who are forced to engage in tedious tasks such as setting up poorly documented and barely usable software for calculations. Often such software was created by theorists and was never designed to be usable by third parties. (I should say immediately that this is not an attack from the Platonic realm; I have created such half-baked software myself and admit my guilt.)

It has often been observed that the level of complexity in this sort of software setup task is much too high. Such software is meant encapsulate a certain amount of expert knowledge and needs to be effective in at least the following ways.

- The build process should be very easy and standardized.
- It should not require a user who is a problem-domain expert.
- It should export good abstractions, and at the right level of abstraction.
- It should be an *open* system.
- The time investment required to produce results should be minimum.

If the software can successfully give the non-expert user a running start, then it can significantly reduce the human energy devoted to this sort of task. The danger is that the created software systems tend to be monolithic, inextensible, and difficult to integrate with other systems. This has been the case historically. Note that monolithic systems are often difficult to test as well, since they are hard to instrument and there is no interface to parameters at intermediate levels.

2 What Can Be Done?

First, much would be accomplished if software producers in this area followed some standard software paradigms. For instance, the issues listed above could be addressed by the following.

- Use a standard build environment; GNU-autoconf-automake is far from perfect, but should be acceptable in most cases.

- Provide *libraries* with clearly documented *interfaces* for manipulating the elements of the problem domain. Do not just create code which is run to produce a result, because it is never clear how to modify or extend such programs or what the dependency relations are between components (assuming that identifiable components can be found at all). Furthermore, well-designed component libraries can be used together, in ways that were not necessarily anticipated by the authors.
- Provide good documentation at all levels of usage.
- Expend effort on the design of the system; the time spent on software design is typically insufficient; software is *not* trivial.
- Design for extensibility from the start. Ask yourself what you might like to do in the future.
- Provide test cases and examples. Provide a clear route to some form of result, even if it is not the most general result obtainable with the tool.

If the producer is a large scientific software “house”, such as a computational division at a national laboratory, they are usually aware of these issues (although awareness does not guarantee success). However, the typical producer, often a theorist who creates the software in order to compute results for a research paper, does not have experience with or interest in creating good software. This is unfortunate and will change only if such producers change their culture. Later, I discuss why this culture change is important and desirable from the viewpoint of research communication.

3 What More Can Be Done?

The problems of flexibility and extensibility in scientific software systems have been addressed directly in recent times. Work in some communities has evolved a paradigm, wherein calculations are controlled through scripting language interfaces. I think this approach is very promising, and I will describe it in some detail below. It is worth saying at the outset that the idea of scripting in special-purpose application domains is not new; what is new is the availability of generic, simple, and well-documented scripting language environments with which to implement such systems.

We can operationally define a scripting language to be a language which is very high-level, is interpreted or possibly byte-compiled, supports late-time binding, is platform-independent from the user's point of view, is extensible, and typically comes with a large library of useful tools and abstractions, including abstractions over standard system services. One speaks of the scripting language *environment* when considering the language together with the rules for building extensions, the conventions about library components, etc. As examples, both Perl and Python satisfy this definition.

In the scientific computing community, Python is rapidly becoming the scripting language of choice because of its simple and consistent design. Without further discussion, and when specifics of the scripting environment are required, I will restrict attention to Python.

Scripting has several immediate benefits.

- Because the scripting environment requires well-defined specification of the interfaces, the interfaces must be thought-out.
- It can be easy to make components work together, avoiding bulky IO code which might otherwise be required. Recall that IO code is often messy and confusing, and it typically requires tedious error checking, which is itself prone to error.
- Users can compose different components together easily and rapidly. In this way they can often tell very early what is going to work and what is not.

The first point is not often emphasized in software circles because the importance of good interfaces is fairly well understood there. However, I think it may be the key point when discussing how to improve tools created by scientists.

4 The SPF Framework

Off and on over the last two years, I have done some work on a system for use in a relatively restricted problem domain. This is the domain of supersymmetric models, with explicit focus on cosmological predictions. This is a problem domain where I have some experience and expert knowledge. At various times I have attempted to create software which carries some of that expert knowledge. The historical source of this effort was a small collection of private codes, created in the old paradigm, which were not initially intended to be publicly released.

As such, they suffered from the usual diseases of inflexibility and obscurity. The code was hard to build, hard to use, could not be used outside the intended modes of operation, and was released as `Neutdriver 1.x`. Eventually I decided to simplify the code base by removing several inflexible “features”, i.e. the ability to compute in certain restricted model types. The resulting code was released as `Neutdriver 2.x`, and is the currently extant version. The latest version includes a GNU-autoconf based build, which probably does more to decrease the usage barrier than any other change I could have implemented.

The new system which I am in the process of creating, called `SPF`, does not extend the physics capabilities of `Neutdriver` at all, but is instead a complete redesign, with explicit concern for usability and flexibility. It is based on the simple idea that only users really know what it is that they want to do, and the best thing that I can do is to create a set of useful components and a system for sewing them together.

The central modules labelled Accelerator Physics and Detection Methods are supposed to be illustrative of a set of specific computational modules, which can be loaded as needed. The module architecture is intended to be simple enough that anyone interested in creating a special purpose module can do so, once in possession of a baseline ability to create a small library, either in the scripting language itself or in a compiled language, say in C or C++. Such modules should be easy to distribute as well, so that users can share what they create.

On the other hand, users who are not interested in creating their own modules need not be exposed to those issues. They can simply choose among the provided standard modules and build computations of interest by writing small scripts. Such small scripts would be the semantic equivalent of the pseudo-code below.

```
load module_for_model_specification
load module_for_computing_result_1
load module_for_computing_result_2
...
parameter_1 = 1.0
parameter_2 = 5.0
for parameter_3 in (1.0 .. 10.0)
    m = create_model(parameter_1, parameter_2, parameter_3)
    result_1 = compute_result_1(m)
    result_2 = compute_result_2(m)
    ...
print result_1, result_2
```

end

Almost anything which depends on specification of a model instance could be a module. For example, it should be fairly easy to take current Monte Carlo event generators, provide them with a scripting language interface which accepts model specification data, and thus make them available for use along with other components.

Furthermore, one can equally well imagine modules for handling post-processing of data. Post-processing is even more likely to be user-dependent than the direct calculational components, since each user is interested in something different. One then imagines something like the following,

```
load my_module_for_post_processing
load module_for_model_specification
load module_for_computing_result_1
load module_for_computing_result_2
...
parameter_1 = 1.0
parameter_2 = 5.0
for parameter_3 in (1.0 .. 10.0)
  m = create_model(parameter_1, parameter_2, parameter_3)
  result_1 = compute_result_1(m)
  result_2 = compute_result_2(m)
  ...
  my_post_processing(result_1, result_2)
end
```

where `my_module_for_post_processing` is a locally created and maintained module for reducing the output in some fashion without creating prohibitively large intermediate data sets.

5 Why Bother?

Many physicists have sympathy for creating a foundational system as described above. However, I think it is fair to say that the revolution has not yet begun. Most theorists or small experimental groups engaged in this sort of exploratory “plot-making physics” need to produce results *right now* and do not feel that they have the time to think about the problems in an abstract way. Most theorists who

produce software which winds up in the hands of the needy show little regard for usability, flexibility, or other “software” issues; of course, their position can be justified by the practical observation that for the most part they have never learned much about these issues and would not know how to proceed.

So where do we find the motivation for changing the culture, in order to break out of this metastable state? I think this motivation lies in the will to communicate effectively. Whatever one feels about the situation, “publish or perish” remains the driving evolutionary pressure in the world of ideas. Ideas which are not properly communicated and used by others are doomed, the authors of the ideas presumably doomed along with them. Any tool which increases the efficiency of communication must become indispensable in this market of ideas.

Consider two papers representing (perhaps only slightly) different models or methods for calculating in certain models. Both contain some theoretical description, at a level of detail which makes it possible (though usually not easy) for a sufficiently motivated student to reproduce the results. Both present some partial results, say in the form of a collection of plots at the end of the paper. Which of these has the greater chance to become important and propagate itself? I think it is clear that the winning idea is the one which is able to make itself more user-friendly to the community of consumers. User-friendliness here includes clear textual discussion, clear mathematical discussion, and the ability to empower the non-expert reader, providing a short and quick path to results.

In many cases, software can be an excellent tool to empower the reader. If I want to see what happens when some parameter λ equals 5.0, but the authors provided partial results, say only for $\lambda = 1.0$, I am typically stuck. There is no way to turn the λ dial on a plot in the back of the paper. Even if one can digest the discussion in the text and thereby make some guess as to what happens, the loss of immediate feedback is usually crippling. The digestion process takes time, ambiguities are usually unavoidable, and nagging doubts impede progress (is that factor of 2 in equation 3.1 a typo or not?). However, if the author provided, instead of the plots at the back of the paper, software for creating the plots which was sufficiently usable, I could generate the plots I wanted in short order and have immediate feedback about my own thoughts. In this way, the software becomes a kind of inexpensive and lightweight carrier of the expert-knowledge of the author.

Another important issue for scientific software is the effectiveness of the review process. The review process not only formalizes acceptance of results but also formalizes the process of understanding. In principle this should force authors to create understandable and usable computational tools. In practice authors rarely go out of their way to do so, and reviewers throw up their hands in despair

when faced with results which depend on some cryptic software, possibly not even made available to the reviewer or the public.

The resistance to creating good tools fills the spectrum from logical and practical argument to fear and greed. Possible responses include

- “That sounds great. What should I do?”
- “I don’t have time for that. It’s not my job. I’m not interested.”
- “But then you would find all the mistakes I made.”
- “It’s not ready yet.”
- “It’s mine. I write papers with it. You can’t have it.”
- “It’s a secret.”

I have, in one form or another, heard all of these responses. Though amusing at first, the phenomenon eventually becomes tedious.

Convincing authors to take seriously their role as tool-producers is clearly a difficult problem which will not be solved by technical means. However, it is possible that if we create a usable infrastructure, or even just a good model for such tools, progress will be measurably accelerated.

A friend once described a behaviour he called “flag-throwing”. Flag-throwing is when, instead of climbing the mountain to plant the flag, authors stand at the base of the mountain and try to throw the flag onto the summit. Since being first to the top is all-important, and there is little overhead in trying a couple times to throw the flag, it is a common practice; sometimes it succeeds. I think that, in a very real sense, a research paper which does not empower the reader by providing useful tools reduces to a kind of flag-throwing activity. Perhaps it is an effective tactic in an unfriendly world. But if we accept that, then perhaps it is time to give up our fantasies about research communication once and for all.